6

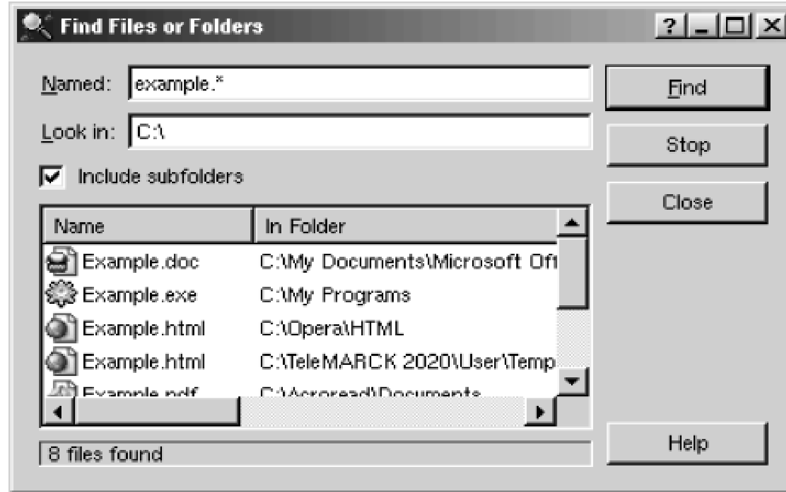
- *Temel dizimler*
- *Ayırıcılar*
- *Alet yığınları*
- *Kaydırma Çubuklu Aletler*
- *Seyyar Pencereler*
- *MDI (Multiple Document Interface)*

Dizim Mekanizmaları

Form üzerine yerleştirilen her bir aletin makul bir ebatı ve pozisyonu olmalıdır. Bazı büyük aletlerin kullanıcının bu aletin içeriğinin tamamını görmesi için kaydırma çubukları olması gerekir. Bu bölümde aletlerin form üzerine yerleştirilme metodları ile demirlenebilir (dockable) pencereler ile MDI pencerelerinin nasıl oluşturulacağını göreceğiz.

Temel Dizimler

Qt çocuk aletlerin form üzerinde tertip edilmesi için üç temel yol tedarik eder: mutlak (absolute) yerleştirme, yedevi (manual) dizim ve dizim mekanizmalar. Şekil 6.1 te görünen Dosya Ara (Finf File) diyaloğunu örnek alarak bu yaklaşımların her birini gözden geçireceğiz.



Şekil 6.1: Dosya Ara diyaloğu

Mutlak yerleştirme aletleri tertib etmenin en kaba yoludur. Bo yöntemde formun çocuklarının ebatları ve yerleri kod içerisinde yazılır ve bunlar sabitdirler, hiç değişmezler. Mutlak yerleştirme kullanıldığında FindFileDialog sınıfının yapıcısı şu şekli alır:

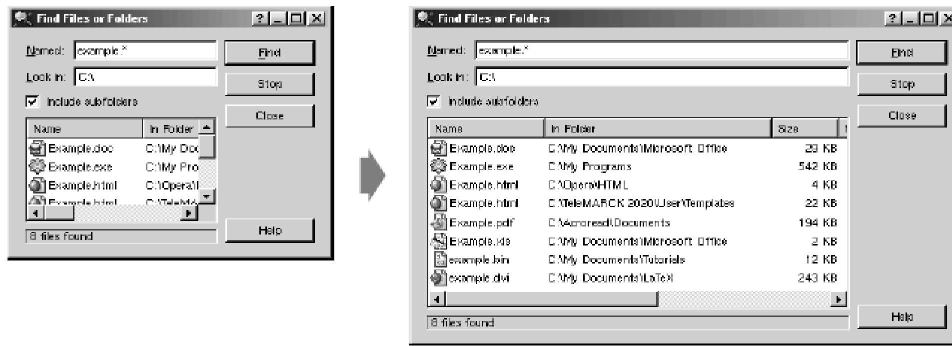
```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
      : QDialog(parent, name)
{
      ...
      namedLabel->setGeometry(10, 10, 50, 20);
      namedLineEdit->setGeometry(70, 10, 200, 20);
      lookInLabel->setGeometry(10, 35, 50, 20);
      lookInLineEdit->setGeometry(70, 35, 200, 20);
      subfoldersCheckBox->setGeometry(10, 60, 260, 20);
      listView->setGeometry(10, 85, 260, 100);
      messageLabel->setGeometry(10, 190, 260, 20);
      findButton->setGeometry(275, 10, 80, 25);
      stopButton->setGeometry(275, 40, 80, 25);
      closeButton->setGeometry(275, 70, 80, 25);
      helpButton->setGeometry(275, 185, 80, 25);
      setFixedSize(365, 220);
}
```

Mutlak yerleştirmanin bir çok avantajı vardır. Dezabantajları arasında en önemlisi kullanıcının pencerenin ebatını değiştirmesine izin verilmesidir. Diğer bir problem ise kullanıcının font boyutunu çok büyük seçmesi durumunda veya proğramın baika dillere çevrilmesi durumunda düğme metni gibi metinlerin bir kısmının gğrüntülenmemeleridir. Bu yaklaşım çok miktarda pozisyon ve ebat hesaplamları yapmamızı gerektirmektedir. Mtlak yerleştirmeye bir alternatifde yedevi yerleştirmedir. Yedevi yerleştirmede aletlerin ebatları ve pozisyonları belirtilmekle birlikte bunlar sabit olmak yerine pencerenin boyutuna göre değişirler. Formun çocuklarının ebatlarını ve pozisyonlarını belirlemesi için resizeEvent() fonksiyonunu yeniden tanımlamak suretiyle bu başarılabilir:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
      : QDialog(parent, name)
{
      ...
      setMinimumSize(215, 170);
      resize(365, 220);
}
void FindFileDialog::resizeEvent(QResizeEvent *)
{
      int extraWidth = width() - minimumWidth();
      int extraHeight = height() - minimumHeight();
      namedLabel->setGeometry(10, 10, 50, 20);
      namedLineEdit->setGeometry(70, 10, 50 + extraWidth, 20);
      lookInLabel->setGeometry(10, 35, 50, 20);
      lookInLineEdit->setGeometry(70, 35, 50 + extraWidth, 20);
      subfoldersCheckBox->setGeometry(10, 60,
                             110 + extraWidth, 20);
      listView->setGeometry(10, 85, 110 + extraWidth,
                             50 + extraHeight);
      messageLabel->setGeometry(10, 140 + extraHeight,
                             110 + extraWidth, 20);
      findButton->setGeometry(125 + extraWidth, 10, 80, 25);
      stopButton->setGeometry(125 + extraWidth, 40, 80, 25);
      closeButton->setGeometry(125 + extraWidth, 70, 80, 25);
      helpButton->setGeometry(125 + extraWidth,
```

```
                                  135 + extraHeight, 80, 25);
}
```

Formun minimum boyutunu 215 × 170 ve ilk boyutunu ise 365 × 220 olarak FindFileDialog sınıfının yapıcısı içerisinde belirledik. resizeEvent() fonksiyonunda ekstar alanı büyütmek istediğimiz alet için kullanıyoruz. Mutlak yerleştirmede olduğu gibi yedevi yerleştirmedede proğramcı tarafından heseplanması gerek çok sayıda sabit değer mevcuttur. Özellikle tasarımın değişmesi halinde bu şekilde kod yazmak gayet yorucudur. Buna ilaveten metinlerin kısmen görüntülnmeme ihtimalede mevcuttur. Aletin çıcuklarının ideal ebatı göz önüne alınarak bu risk ortadan kaldırılabilir ancak bu proğramın gayet karmaşık olmasına yol açar.



Şekil 6.2: Ebeatı değiştirilebilen bir diyaloğun büyütülmes (küçültülmesi).

Aletlerin bir form üzerine yerleştirilmesi amacıyla kullanılacak en müğsait yol Qt nin dizim mekanizmalarını kullanmaktıur. Dizim mekanizmaları her tür alet için makul ayarlar tedarik etmekle birlikte aletlerin, font büyüklüğü, türü ve içeriklerine bağlı olarak değişen ideal boyutlarını göz önünde bulundururlar. Dizim mekanizmaları aletlerin minimum ve maksimum ebatlarını göz önünde bulundurdukları gibi otomatik bir şekilde fontun değişmesi, metinlerin değişmesi ve pencerenin ebatının değiştirilmesi durumunda alet dizimlerini yeniden ayarlarlar. Qt üç tane dizim mekanizması tedarik eder: QHBoxLayout, QVBoxLayout ve QGridLayout. Bu sınıflar dizim mekanizmalarının temelini teşkil eden QLayout sınıfnının varısleridirler. Bu üç sınıftan her biri Qt Designer tarafından tamamen desteklenmemektedir ve aynı zamanda Qt Designer dışında kodlamada direk olarakta kullanılabilirler. İkinci bölümde her iki yaklaşımıda kullanan misaller mevcuttur. İşte FindFileDialog sınıfının dizim mekanizmalarını kullanan kodu:

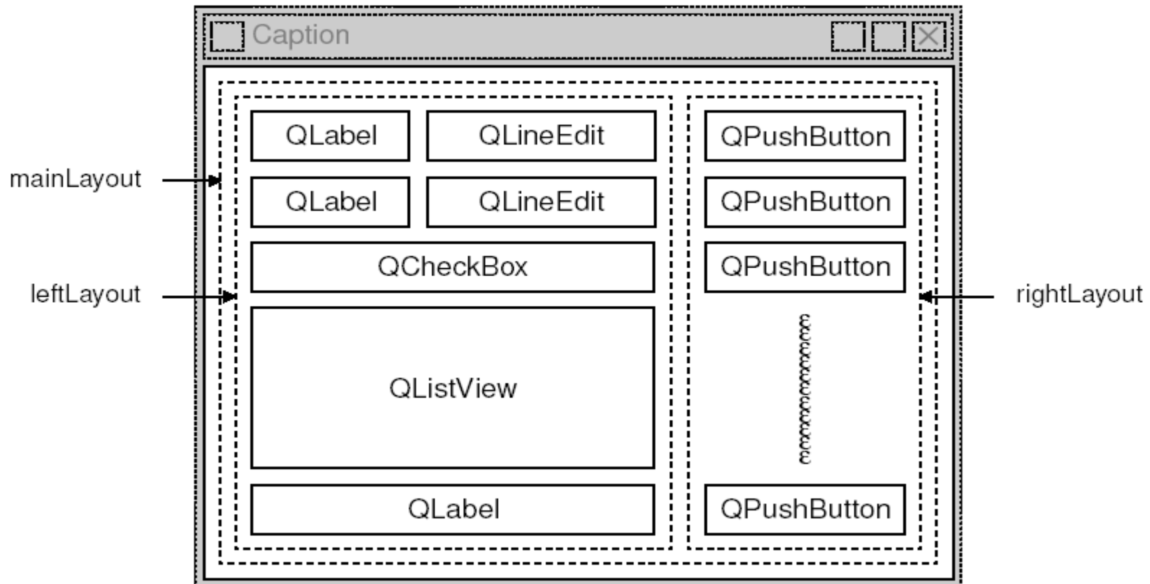
```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
:QDialog(parent, name)
{
      ...
      QGridLayout *leftLayout = new QGridLayout;
      leftLayout->addWidget(namedLabel, 0, 0);
      leftLayout->addWidget(namedLineEdit, 0, 1);
      leftLayout->addWidget(lookInLabel, 1, 0);
      leftLayout->addWidget(lookInLineEdit, 1, 1);
      leftLayout->addMultiCellWidget(subfoldersCheckBox,2,2,0,1);
```

```
        leftLayout->addMultiCellWidget(listView, 3, 3, 0, 1);
        leftLayout->addMultiCellWidget(messageLabel, 4, 4, 0, 1);

        QVBoxLayout *rightLayout = new QVBoxLayout;
        rightLayout->addWidget(findButton);
        rightLayout->addWidget(stopButton);
        rightLayout->addWidget(closeButton);
        rightLayout->addStretch(1);
        rightLayout->addWidget(helpButton);

        QHBoxLayout *mainLayout = new QHBoxLayout(this);
        mainLayout->setMargin(11);
        mainLayout->setSpacing(6);
        mainLayout->addLayout(leftLayout);
        mainLayout->addLayout(rightLayout);
}
```

Tertib yada dizim işlemi bir QHBoxLayout, bir QgridLayout ve birde QVBoxLayout tarafından yapılmaktadır. Solda QgridLayout (leftLayout) ve sağda QVBoxLayout (rıghtLayout) yan yana yerleştirilmişlerdir ve her ikisiside QHBoxLayout (mainLayout). Diyaloğun etrafındaki haşiye yada marjin 11 piksel genişliginde ve çocuk aletler arasındaki bşluk ise 6 piksel genişliğindedir.



Şekil 6.3: Dosya Ara (Find File) diyaloğunun tertibi.

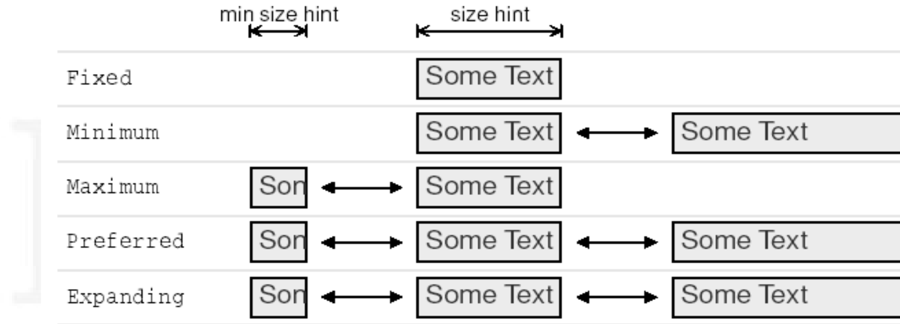
QGridLayout iki boyutlu ızgara şeklindeki hücrelerden ibarettir. Sol üst köşedeki QLabel (0, 0) poziyonunda (yada hücresinde) ve mukabili olan QLineEdit ise (0, 1) pozisyonuda yer alırlar. QCheckBox iki kolunu birden kaplamaktadır; bu alan (2, 0) ve (2, 1) hücre pozisyonlarından ibarettir. Onun altındaki QListView ve QLabel da iki kolona yayılmışlardır. addMultiCellWidget() fonksiyonu şu şekilde çağrılır:

```
leftLayout->addMultiCellWidget(widget, row1, row2, col1, col2);
```

ki burada widget dizime eklenecek olan alet, (row1, col1) ise alet tarafından istila edilecek olan sol üst köşe ve (row2, col2) ise aletin istia edeceği sağ alt köşedir. Aynı diyalog Qt Designer altında şu şekilde oluşturulabilir: Önce çıcuklar takribi yerlerine yerkeştirilir ve sonra birlikte yerleştirilmesi gereken aletler seçildikten sonra Layout|Lay Out Horizontally, Layout|Lay Out Vertically, yada Layout|Lay Out in a Grid menülerinden uygun olanı seçilir. İkinci bölümde bu yaklalşımı kullanarak Spreadsheet proğramının Hücreye Git ve Tasnif Et diyaloglarını oluşturuken kullandık. Dizim mekanizmalarının şu ana kadar gördüğümüz faidelerine ilaveten başka avantajlarıda mevcuttur. Bir dizime bir alet eklenmesi yada bir aletin ondan silinmesi durumunda dizim mekanizması otomatik olarak içeriğini tertib eder. Bu bir aletin hide() ve show() fonksiyonu ile saklanıp gösterilmesi durumu içinde geçerlidir. Bir aletin ideal ebatının değişmesi durumunda dizim mekanizması dizimi yeniden tertib eder ve bunu yaparken aletin yeni ideal boyutu göz öňğne alınır. Yine dizim mekanizmaları bir formun minimum ebatını onun ihtiva ettiği aletlerin minimum ebatları ve ideal ebatlarını göz öňğnde bulundurarak otomatik olarak ayarlar. Şu ana kadar gördüğümüz misallerin hepsinde aletleri dizim içerisinde yerleştirdikten sonra boşlukları kaplaması için aralıkçılar (spacer items) kullandık. Bazan bu yaklaşım aletleri arzu ettiğimiz şekilde yerleştirmemiz için kafi gelmez. Böyle durumlarda aletlerin ideal boyutlarını ve ebat değiştirme tarzlarını (size policies) değiştirmek suretiyle arzu ettiğimiz tertibi yada dizimi elde ederiz. Bir aletin ebat değiştirme tarzı dizim mekanızmasına onun nasıl büyüyüp küçülmek istediğini bildirir. Qt aletlerinin her biri için makul ebat değiştirme tarzları tedarik etmektedir ancak bu tarzların bütün ortamlarda istenen sonucu vermesi beklenemez bu bakımdan proğramcının zaman zaman bunu form tasarımına göre ayarlaması gerekir. Ebat değiştirme tarzının bir düşey ve birde yatay bileşeni vardır. Her bir bileşen için en kullanışlı değerler şunlardır: Fixed, Minimum, Maximum, Preferred, and Expanding:

- Fixed, aletin büyüyüp yada küçülemyeceği anlamına gelir. Alet daima ideal ebatını muhafaza eder.
- Minimum, aletin ideal ebatının onun minimum ebatı olduğunu bekirler. Alet ideal ebatından daha küçük yapılamaz ancak gerkirse boş kalan alnmı doldurmak için büyüyebilir.
- Maximum, aletin ideal ebatının onun maximum ebatı olduğunu anlamına gelir. Alet minimum boyutuna kadar küçültülebilir.
- Preferred, aletin ideal boyutunu tercih ettiği ancak gerekirse büyüyüp küçülebileceği anlamına gelir.
- Expanding, aletin hem büyüyebileceği hemde küçülebileceği ancak daha ziyade büyümeyü tercih ettiği anlamına gelir.

Şekil 6.4 de farklı ebat değiştirme tarzları “Some Text” metnini görüntüleyen bir QLabel örneği ile gösterilmektedir.



Şekil 6.4: Değişik ebat değiştirme tarzları.

Preferred ve Expanding ebat değiştirme tarzlarına sahip iki aleti ihtiva edeb bir formun ebat değiştirmesi durumunda ekstra boşluk Expanding tarzına sahip olana alet verilir ve Preferred tarzına sahip alet ebatını muhafaza eder.

İki tane daha ebat değiştiröe tarzi mevcuttur: MinimumExpanding ve Ignored. Bunlardan ilki bazı nadir durumlarda eski Qt versiyonlarında gerekli idi ancak artık buna ihtiyaç yoktur; daha makul bir yaklaşım Expanding tarzını kullanıp aletin minimumSizeHint() fonksiyonunu yeniden tanımlamaktır. İkincisi yani Ignored ise Expanding tarzına benzemekle birlikte aletin ideal botyrunu gözardı eder.

Ebat değiştirme tarzlarının düşey ve yatay bileşenlerine ilaveten QSizePolicy sınıfı aynı zmanada düşey ve yatay gerilme faktörlerinide (stretch factor) muhafaza eder. Bu faktörler dizim mekanizmalarını farkli aletlerin formun büyümesi durumunda farklı hızlarla büyüyeceklerini bidirmek maksadı ile kullanılabilirler. Mesela, bir QListView aleti bir diğer QTextEdit aletinin üzerinde yer alıyorsa ve biz QTextEdit altinin yüksekliğinin QlistView aletininkinin iki katı olmasını istiyorsak, QTextEdit aletinin düşey gerilme faktörünü (stretch factor) 2 ve QListView düşey gerilme faktörünü ise 1 yaparız

Dizimi etkilemenin bir diğer yoluda çocuk aletleri için bir minimum ebat, bir maximum ebat yada bir sabit ebat (fixed size) kullanmaktır. Dizim mekanizması aletleri tertib ederken bu liöitleri göz önünde bulundurur. Buda yeterli değil ise, çocuk aletin bir alt sınıfını oluşturur ve onun sizeHint() fonksiyonun ihtiyacımız olan ideal ebatı elde etmek için yeniden tanımlarız.

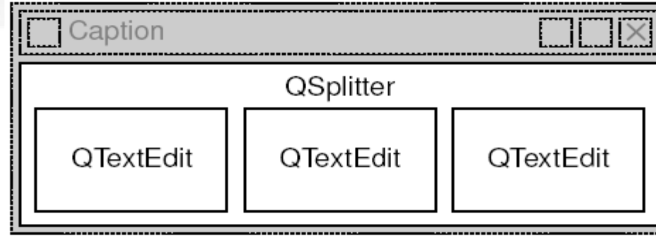
Bölücüler (Splitters)

Bölücü bir alet olup diğer aletleri ihtiva eder ve bu aletleri birbirlerinden kulplar (hanles) ile ayırır. Kullanıcı bu kulpları kaydırnmak suretiyle çocuk aletlerin ebatlarını değiştirebilir. Bölücüler dizim mekanizmalarına alternatif olarak kullanıcıya çocuk aletlerin boyutlarını değiştirnesi içim daha fazla kontrol vermek için kullanlabilirler. Qt içerisinde bölücüler QSplitter aletini kullanarak geröekleştirilirler. QSplitter aletinin çocukları oluşturulma sıralarına göre ya yan yana yada alt alta aralarında ayırma çubukları olacak şekilde yerleştirilirler. İşte şekil 6.5 de izhae edilen pencerenin oluşturulmasında kullanılan kod:

```
#include <qapplication.h>
```

```
#include <qsplitter.h>
#include <qtextedit.h>
int main(int argc, char *argv[])
{
      QApplication app(argc, argv);
      QSplitter splitter(Qt::Horizontal);
      splitter.setCaption(QObject::tr("Splitter"));
      app.setMainWidget(&splitter);
      QTextEdit *firstEditor = new QTextEdit(&splitter);
      QTextEdit *secondEditor = new QTextEdit(&splitter);
      QTextEdit *thirdEditor = new QTextEdit(&splitter);
      splitter.show();
      return app.exec();
}
```

Bu misal yan yana QSplitter aleti içerisine yerleştirilmiş olan üç adet QTextEdits aletinden ibarettir. Görevleri aletleri tertib etmekten ibaret olan dizim mekanizmalarının aksine QSplitter aleti QWidget aletinin varisidir ve herhangi bir alet gibi kullanılabilir.



Şekil 6.5: Bölücü (Qsplitter) aletinin kullanımı.

QSplitter çocuklarını ya düşey yada yatay bir şekilde tertib eder. Karmaşık dizaynları oluşturmak için mütedahil düşey ve yatay bölücü[1] (QSplitter) aletleri kullanılabilir. Mesel, şekil 6.6 da gösterilen Mail Clientproğramı yatay bir QSplitter ve onun sağında yer alan düşey bir QSplitter eder. İşte Mail Client proğramının QMainWindow altsınıfina ait kod:

```
MailClient::MailClient(QWidget *parent, const char *name)
      : QMainWindow(parent, name)
{
      horizontalSplitter = new QSplitter(Horizontal, this);
      setCentralWidget(horizontalSplitter);

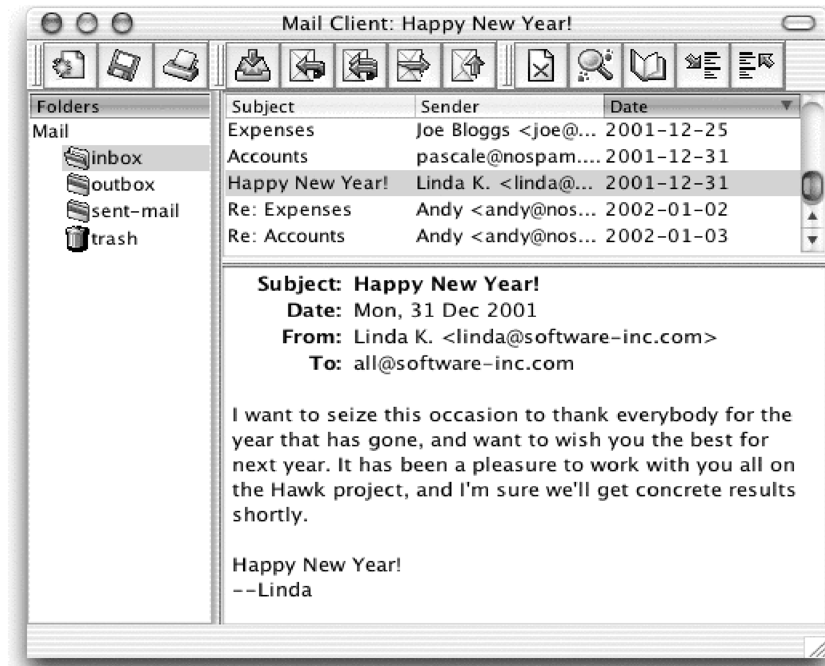
      foldersListView = new QListView(horizontalSplitter);
      foldersListView->addColumn(tr("Folders"));
      foldersListView->setResizeMode(QListView::AllColumns);
      verticalSplitter = new QSplitter(Vertical,
                                       horizontalSplitter);
      messagesListView = new QListView(verticalSplitter);
```

1 Yatay ve düşey bölücüden maksat çocuklarını sırasıyla yan yana ve alta dizen bölücüdür.

```
        messagesListView->addColumn(tr("Subject"));
        messagesListView->addColumn(tr("Sender"));
        messagesListView->addColumn(tr("Date"));
        messagesListView->setAllColumnsShowFocus(true);
        messagesListView->setShowSortIndicator(true);
        messagesListView->setResizeMode(QListView::AllColumns);

        textEdit = new QTextEdit(verticalSplitter);
        textEdit->setReadOnly(true);
        horizontalSplitter->setResizeMode(foldersListView,
        QSplitter::KeepSize); verticalSplitter->setResizeMode(
                    messagesListView, Qsplitter::KeepSize);
        ...
        readSettings();
}
```

Önce yatay bölücüyü (QSplitter) oluşturup onu ana pencerenin (QmainWindow) merkezi aleti olarak tesbit ediyoruz. Daha sonra çocuk aletleri ve onlarında, varsa, çocuklarını oluşturuyoruz.



Şekil 6.6: Mac OS X altında çalışan "Mail Client" isimli bir e-mail proğramı.

Kullanıcı pencerenine ebatını değiştirdiğinde, QSplitter yeni açığa çıkan alanı çocuk aletler arasında eşit olarak taksim eder taki çocukların nisbi (bir birlerine oranla) ebatları aynı klasın. Mail Client emisalinde biz bu davranışı istemiyoruz, bilakis iki QlistView aletlerinin ebatlarını muhafaza ederken ziyade alnın QtextEdit tarafından kullanılmasını tercih ediyoruz. Bunu setResizeMode() fonksiyonuna yaptığımız iki çağrı ile başarıyoruz. Proğram

başlatıldığında, QSplitter aletlere onları normal bir şekilde boyutlandırır. Qsplitter::setSizes() fonksiyonunu çağırmak suretiyle bölücünün kulpunu kod içerisinde kaydırabilriz. QSplitter sınıfı aynı zamanda onun vaziyetinin mufaza edilip proğram daha sonra çalıştırıldığında o vaziyete geri dönmesine imkan sağlar. İşte Mail Client ptoğramının ayarlarını kaydeden writeSettings()[2] fonksiyonu:

```
void MailClient::writeSettings()
{
      QSettings settings;
      settings.setPath("software-inc.com", "MailClient");
      settings.beginGroup("/MailClient");
      QString str;
      QTextOStream out1(&str);
      out1 << *horizontalSplitter;
      settings.writeEntry("/horizontalSplitter", str);
      QTextOStream out2(&str);
      out2 << *verticalSplitter;
      settings.writeEntry("/verticalSplitter", str);
      settings.endGroup();
}
```

İşte writeSettings() fonksiyonunun mukabili olan readSettings()[3] fonksiyonu:

```
void MailClient::readSettings()
{
      QSettings settings;
      settings.setPath("software-inc.com", "MailClient");
      settings.beginGroup("/MailClient");
      QString str1 = settings.readEntry("/horizontalSplitter");
      QTextIStream in1(&str1); in1 >> *horizontalSplitter;
      QString str2 = settings.readEntry("/verticalSplitter");
      QTextIStream in2(&str2); in2 >> *verticalSplitter;
      settings.endGroup();
}
```

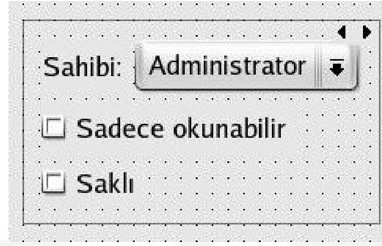
Bu fonksiyonular QTextStream alt sınıfları olab QTextIStream ve QtextOStream sınıflarına dayanırlar. Normalde nölücü kulpu kaydırılma esnasında ince bir çizgi olarak görüntülenir ve bölücünün her iki tarafındaki aletler kullanıcı farenin tuşunu bıraktıktan sonra güncelleştirilirler. QSplitter dan çocuklarını kullanıcının bölücü kulpunu kaydırması esnasında güncelleştirmeisni istiyor isek bu durumda setOpaqueResize(true) fonksiyonunu çapırırız. QSplitter aleti Qt Designer altında kullanılabilir. Aletleri bir bölücü içine yerleştirmek için önce aletler arzu edilen pozisyona yerleştirilir ve daha sonra ya Layout|Lay Out Horizontally (in Splitter) veya Layout|Lay Out Vertically (in Splitter) menü komutları seçilir.

2 ayrarları kaydet

3 ayarları oku

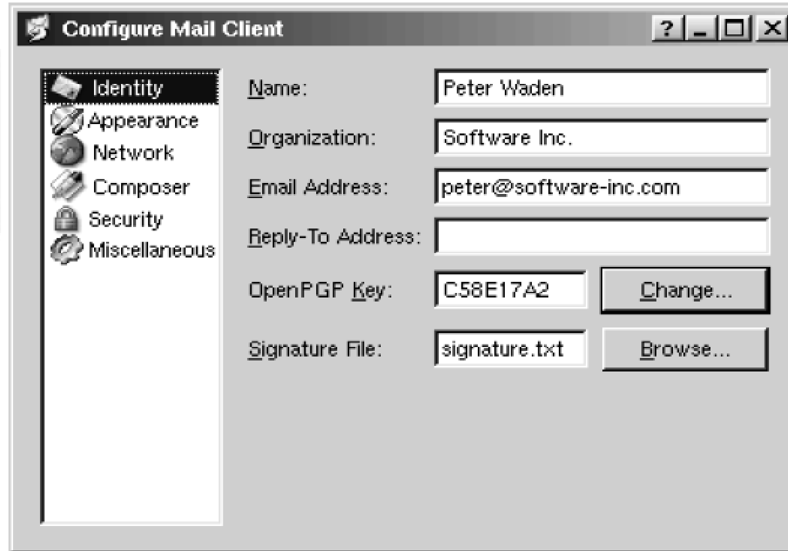
Alet Yığınları (Widget Stacks)

Aletleri tertip edilmesi için faydalı olan bir aletde QWidgetStack sınıfıdr. Bu alet bir çok çocuk alet veya sahife ihtiva edebilir ancak bunlardan yanlızca bir tanesini görüntüleyip geriye kalan kısmını kullanıcıdan saklar. Yığın içerisinde sahifeler sıfırdan başlayarak numaralandırılmışlardır. Belirli bir çocuk aleti görüntülemek istediğimizde raiseWidget() fonksiyonunu ya sahife numarasını kullanarak yada aletin bir müşirini kullanarak öağırırız.



Şekil 6.7: QWidgetStack.

QWidgetStack aleti kullanıcı tarafından görülemez ve kullanıcı sahife değiştiremez. Şekil 6.7 de gösterilen koyu rankli çerçeve ve sağ üst köşedeki oklar Qt Designer tarafından QWidgetStack aletininin tasarımı esnasında kolaylık olması anacıyla tedarşk edilmişlerdir.



Şekil 6.8: Mail Client proğramının Ayarla (Configure) diyaloğu.

Şekil 6.8 de gçsterilen Ayarla dialoğu QWidgetStack kullanır. Bu diyalog sol tarafta bir

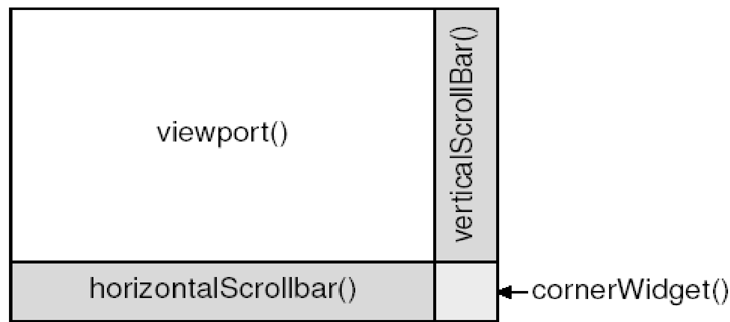
QListBox ve sağ tarfta ise bir QWidgetStack ihtiva eder. QListBox içerisindeki her bir ferd (item) QWidgetStack . Bu tür formlar Qt Designer içerisinde kolaylıkla oluşturulabilirler:

1. Bir diyalog yada Alet şablonuna göre bir form oluştur.
2. Forma bir liste kutusu (list box) ve birde alet yğını (widget stack) ekle.
3. Alet yığınının her bir sayfasına çocuklarını dizim mekanizmalarını kullanarak yerleştir(Yeni sayfa oluşturmak için sağ fare tuğuna bas ve Add Page (sayfa ekle) menüsünü seç; syfa değiştirmek için alet yığınının üst sağ köşesindeki küçük oklara tıkla.)
4. Aletleri yatay dizim (horizontal layout) mekanizması kullanarak yan yana yerleştir.
5. Alet kutusunun (list box) highlighted(int) sinyalini alet yığınının (widget stack) raiseWidget(int) dilimine başla.
6. Alet kutusunun currentItem özelliğinin deşerini 0 yap.

Alet yığınında sayfaları değiştirmek için mevcut sinyal ve dilimleri kullandığımız için bu forma Qt Designer içerisinde Preview menüsünü kullanarak gözatabiliriz.

Kaydırma Çubuklu Aletler (Scroll Views)

QScrollView sınıfı kaydırma çubukları olan bir viewport ve bir köşe aleti (corner widget, şekil 6.9) tdarik ederki bu genelde boş bir QWidget dır. Bir alete kaydırma çubukları eklenmesi gerekiyorsa bu durumda QScrollView kullanmak bir alete QScrollBars ekleyip onun düzenli çalışmasını sağlamaktan daha kolaydır.



Şekil 6.9: QScrollView in ihtiva ettiği aletler.

QScrollView kullanmanın en kolay yolu kaydırma çubukları eklemek istediğimiz alet ile addChild() fonksiyonunu çağırmaktır. QScrollView otomatik olarak aletin atasını değiştirir ve onu viewport (QScrollView::viewport()) aletinin çocuğu yapar. Mesela, beşinci bölümde yazdığımız IconEditor aletine kaydırma çubukları şu şekilde ekleyebiliriz:

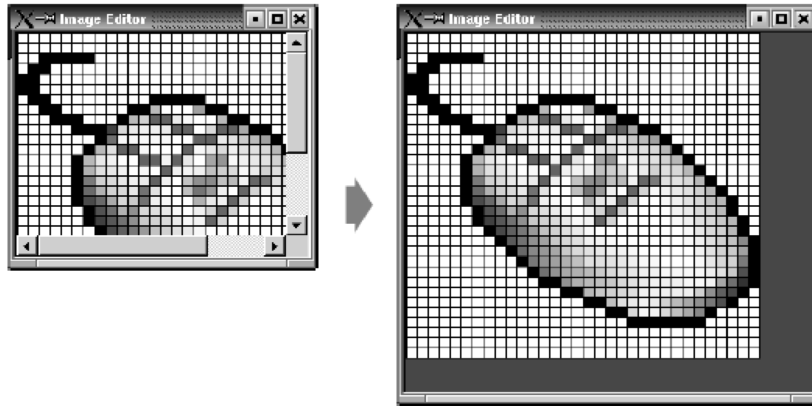
```
#include <qapplication.h>
```

```
#include <qscrollview.h>
#include "iconeditor.h"
int main(int argc, char *argv[])
{
      QApplication app(argc, argv);
      QScrollView scrollView;
      scrollView.setCaption(QObject::tr("Icon Editor"));
      app.setMainWidget(&scrollView);
      IconEditor *iconEditor = new IconEditor;
      scrollView.addChild(iconEditor);
      scrollView.show();
      return app.exec();
}
```

Normalde kaydırma çubukları sadece viewport alanının çocuk aletten daha küçük olması durumunda görüntülenir. Şu kodu yazmak suretiyle katdırma çubuklarının sürekli görünütülenmelerini sağlayabiliriz:

```
scrollView.setHScrollBarMode(QScrollView::AlwaysOn);
scrollView.setVScrollBarMode(QScrollView::AlwaysOn);
```

Çocuk aletin ideal boyutu (size hint) değişince QScrollView otomatik olarak bu değişikliği göz önüne alır.



Şekil 6.10: QScrollView in ebatının değiştirilmesi.

QScrollView aletini kullanmanın bir diğer yoluda QScrollView sınıfının bir alt sınıfını oluşturup aletin içeriğini çizmesi için drawContents() fonksiyonunu yeniden tanımlamaktıe. QIconView, QListBox, QListView, QTable, ve QtextEdit gibi Qt sınıfları bu yaklaşımı kullanırlar. Bir aletin kaydırma çubuklarına ihtiyacının olması ihtimali varsa onun QScrollView sınınfının alt sınıfı yapmak tavsiye olunur. Bunun nasıl yapılabileceğini göstermek için IconEditör sınınıfının yeni bir versiyonun QScrollView sınınfının alt sınıfı olarak oluşturacağız. Yeni sinifa ImageEditor adını vereceğiz çünkü kaydırma çubukları sayesinde yeni sinif büyük resimlerler çalışabilir. ,

```
#ifndef IMAGEEDITOR_H
#define IMAGEEDITOR_H
#include <qimage.h>
#include <qscrollview.h>
class ImageEditor : public QScrollView
{
      Q_OBJECT
      Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
      Q_PROPERTY(QImage image READ image WRITE setImage)
      Q_PROPERTY(int zoomFactor READ zoomFactor WRITE
                  setZoomFactor)
public:
      ImageEditor(QWidget *parent = 0, const char *name = 0);
      void setPenColor(const QColor &newColor);
      QColor penColor() const { return curColor; }
      void setZoomFactor(int newZoom);
      int zoomFactor() const { return zoom; }
      void setImage(const QImage &newImage);
      const QImage &image() const { return curImage; }
protected:
      void contentsMousePressEvent(QMouseEvent *event);
      void contentsMouseMoveEvent(QMouseEvent *event);
      void drawContents(QPainter *painter, int x, int y, int
                        width, int height);
private:
      void drawImagePixel(QPainter *painter, int i, int j);
      void setImagePixel(const QPoint &pos, bool opaque);
      void resizeContents();
      QColor curColor;
      QImage curImage;
      int zoom;
};
#endif
```

Başlık dosyası orjinaline çok benzemektedir(p. 100). Aralarındaki ana fark QWidget yerine QScrollView sınıfının alt sınıfını oluşturmamızdır. Sınıfın geri kalan kısmını oluştururken daha başka farkalılıkların varlığını müşahade edeceğiz.

```
ImageEditor::ImageEditor(QWidget *parent, const char *name)
      :QScrollView(parent, name, WStaticContents | WNoAutoErase)
{
      curColor = black;
      zoom = 8;
      curImage.create(16, 16, 32);
      curImage.fill(qRgba(0, 0, 0, 0));
      curImage.setAlphaBuffer(true);
      resizeContents();
}
```

Yapıcı WStaticContents ve WNoAutoErase seçeneklerini QScrollView sınıfının yapıcısına gçnderiyor. Bu seçenekler aslında QScrollView sınıfının viewport (Şekil 6.9)alanı içindir. Ebat değiştirme tarzına dokunmuyoruz çğnkğ QScrollView sınıfının normal ebat değiştirme tarzı olan (Expanding, Expanding) gayet müsaittir. Bu sınıfın orjinal versiyonunda updateGeometry() fonksiyonunu yapıcı içerisinde çağırmadık bunun sebebise Qt nin dizim

mekanizmalarının aletin ilk ebatını bulabilmeleridir. Ancak burada QScrollView temel sınınfının ilk ebatını vermeliyiz ve bunu resizeContents() fonksiyonunu çağırarak yaparız.

```
void ImageEditor::resizeContents()
{
      QSize size = zoom * curImage.size();
      if (zoom >= 3) size += QSize(1, 1);
      QScrollView::resizeContents(size.width(), size.height());
}
```

resizeContents() fonksiyonu, QscrollView::resizeContents() fonksiyonunun QScrollView in data içeren kısmının ebatı ile çağırır. Kaydırma çubukları data içeren kısımın büyüklüğünün viewport kısmının büyüklüğüne oranına bağlı olarak gösterilir veya gizlenirler. sizeHint() foksiyonunun yeniden tanımlamamıza gerek yoktur; bu fonksiyonun QScrollView içerisindeki orjinal versiyonu içeriğinin büyüklüğüne bağlı olarak makul bir ideal büyüklük tedarik eder.

```
void ImageEditor::setImage(const QImage &newImage)
{
      if (newImage != curImage)
      {
            curImage = newImage.convertDepth(32);
            curImage.detach();
            resizeContents();
            updateContents();
      }
}
```

Orjinal IconEditor fonksiyonlarının çoğunda, aletin yeniden boyanması için update() fonksiyonunu kullanarak randevü aldık ve updateGeometry() fonksiyonunu çağırarak ideal ebat değişikliğinin göz önüne alınmasını sağlarız. Yeni versiyobda bunların yerine updateContents() fonksiyonunu çağırarak QScrollView aletini içeriğin ebatının değiştiğinden haberdar ederiz ve updateContents() fonksiyonunu çağırarak aleti yeniden boyanmaya mecbur kılarız.

```
void ImageEditor::drawContents(QPainter *painter, int, int, int,
int)
{
      if (zoom >= 3)
      {
            painter->setPen(colorGroup().foreground());
            for (int i = 0; i <= curImage.width(); ++i)
                  painter->drawLine(zoom * i, 0, zoom * i, zoom *
                        curImage.height());
            for (int j = 0; j <= curImage.height(); ++j)
                  painter->drawLine(0, zoom * j, zoom *
                        curImage.width(), zoom * j);
      }
      for (int i = 0; i < curImage.width(); ++i)
      {
            for (int j = 0; j < curImage.height(); ++j)
                  drawImagePixel(painter, i, j);
      }
```

```
}
```

drawContents() fonksiyonu QScrollView tarafından içerik kısmının yeniden boyanması için çağrılır. QPainter nesnesi kaydırma çubuklarından dolayı meydana gelecek farkı göz önüne alacak şekilde ayarlanmıştır. paintEvent() fonksiyonu içerisinde boyama işlemini normal bir şekilde yaparız. İkinci, üçüncü, dördüncü ve beşinci argümanlar boyanacak alnı belirlerler. Bu dikdörtgeni sadece boyanması gereken kısma ayarlayabilirdik ancak kolaylık olması bakımından her şeyi boyuyuruz. drawContents() fonksiyonunun sonunda çağrılan drawImagePixel() fonksiyonu aslında original IconEditor (p. 106) sınınfınınki ile aynı olduğundan burada tekrarlamaya gerek duymadık.

```
void ImageEditor::contentsMousePressEvent(QMouseEvent *event)
{
      if (event->button() == LeftButton)
            setImagePixel(event->pos(), true);
      else if (event->button() == RightButton)
            setImagePixel(event->pos(), false);
}
void ImageEditor::contentsMouseMoveEvent(QMouseEvent *event)
{
      if (event->state() & LeftButton)
            setImagePixel(event->pos(), true);
      else if (event->state() & RightButton)
            setImagePixel(event->pos(), false);
}
```

QScrollView aletinin ihtiva eden kısmının fare eylemlerini halletmek için QScrollView içerisindeki hususi eylem halledicileri yeniden tanımlamak gerekir ki bunların isimleri “contents” ile başlar. Perde arkasında, QScrollView otomatik olarak viewport kordinatlarını içerik aletinin kordinatına çevirdiği için bizim herhangi bir çeviri yapmamaıza gerek yok.

```
void ImageEditor::setImagePixel(const QPoint &pos, bool opaque)
{
      int i = pos.x() / zoom;
      int j = pos.y() / zoom;
      if (curImage.rect().contains(i, j))
      {
            if (opaque)
                  curImage.setPixel(i, j, penColor().rgb());
            else
                  curImage.setPixel(i, j, qRgba(0, 0, 0, 0));
            QPainter painter(viewport());
            painter.translate(-contentsX(), -contentsY());
            drawImagePixel(&painter, i, j);
      }
}
```

setImagePixel() fonksiyonu contentsMousePressEvent() ve contentsMouseMoveEvent() tarafından bir pikseli boyamak ve silmek amacıyla çağrılır. Bu fonksiyonun kaynak kodu orjinal versiyonuna çok benzemekle birlikte aralarındaki farklardan birir QPainter nesnesinin oluşturulmasıdır. QPainter nesnesinin atası olarak viewport() kullandık çünkü boyama işlemi viewport üzerinde gerçekleştirilecek. Aynı zamanda kaydırma dolayısıyla

oluşan farkı hesaba katmak için QPainter ın kordinat systemini kaydırıyoruz.

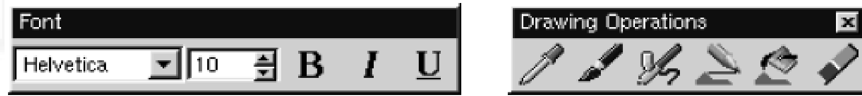
QPainter ile alaklı olan üç satırın yerine çu satırı kullanırız:

```
updateContents(i * zoom, j * zoom, zoom, zoom);
```

Bu QScrollView den sadece büyültülmüş olan rsimin kapladığı dikdörgeni güncelleştirmeisni ister. Ancak drawContents() fonksiyonunu sadece gerekli olan alanı boyayacak şekilde optimize etmediğizden bu verimli olmaz bu yüzden bir QPainter oluşturup boyama işlemini kendimizin yapması daha verimli olur. ImageEditor sınıfını bu haliyle QWidget üzerine kurulmuş olan QScrollView içerisinde yer alan IconEditor den kullanım bakımından nerede ise farksızıdır. Binaenaleyh daha gelişmiş aletlerin oluşturulması için QScrollView dan bir alt sınıf oluşturmak o aleti QScrollView içerisinde kullanmaktan daha tabiidir. Mesela, QTextEdit gibi bir sınıf ki implements wordwrapping bu içerdiği metin ile QScrollView arasında çok sıkı bir bağlantı gerektirir. Yine QScrollView sınıfının iöeriği çok uzun veya geniş olack ise bu durumdada QScrollView dan alt sınıf oluşturulmalıdır çünkü bazi işletim sisyemleri 32,767 pikselden daha fazlasını desteklememektedirler. QScrollView sınınfının ImageEditor misalinde gösterilmeyen bir yönüde viewport alanına çocuk aletlerin yerleştirilebilmeleridir. Çocuk aletler addWidget() fonksiyonunu kullanarak eklenirler ve moveWidget() fonksiyonunu kullanarak silinirler. Kullanıcı içerik kısmını kaydırdığında, QScrollView otomatik olarak çocuk altleri kaydırır.(QScrollView nın çok sayıda çocuğu var ise bu kaydırma işlemini bir hayli yavaşlatabilir bu durumda enableClipper(true) fonksiyonunu çağırarak optimize edebiliriz) Bu yaklaşımın kullanılabileceği proğrama en güzel çrnek web browser dır. İçereğin hememn hemen hepsi vıewport üzerine çizilirken, düğmeler ve diğer girdi elemenları çocuk aletler olarak teşhir edilir.

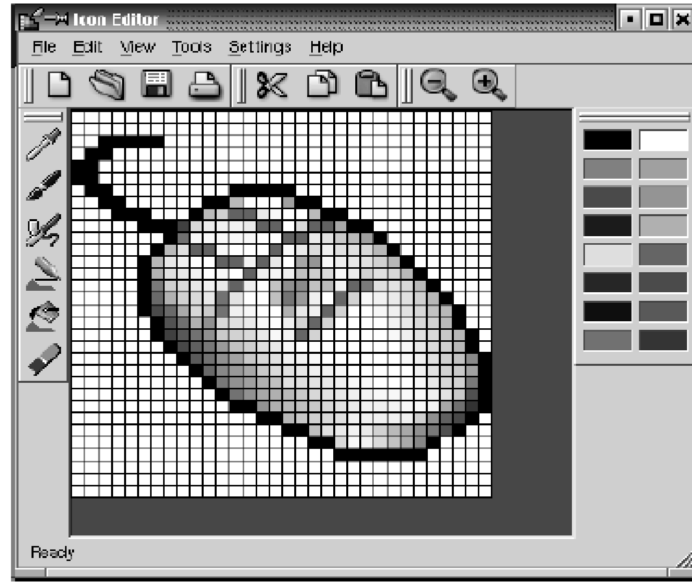
Dock (Seyyar) Pencereleri

Seyyar pencereler "dock" alanına takılabilen pencerelerdir. Alet kutuları (toolbars) seyyar pencerelerin en bariz misallerindendir ancak daha başka türlerde mevcuttur. QMainWindow aleti, merkezi aletin üstünde, altında, sağında ve solunda olmak üzere dört tane dock alnı tedarik eder. QToolBar oluşturduğumuzda o kendisini otomatik olarak atasının ğst dock alanına yerleştirir.



Şekil 6.11: Yüzen seyyar pencereler.

Her seyyar pencerenin bır kulpu vardır. Şekil 6.12 gösterildiği gibi bu ya düşey veyahutta yatay iki çizgiden ibarettir. Kullanıcı seyyar pencereleri bir rıhtım alanından diğer bir rıhtım alanına kulpunu sürüklemek suretiyle aktarabilirler. Bir seyyar pencereyi rıhtım alanlarının dışına sürüklemek suretiyle onu başlı başına yüzen bir pencere haline getirmek mümkündür. Bu şekilde yüzen seyyar pencerelerin kendilerine has başlıkları olduğu gibi pencereyi kapatmak için birde kapat (X) düğmeleride olabilir.

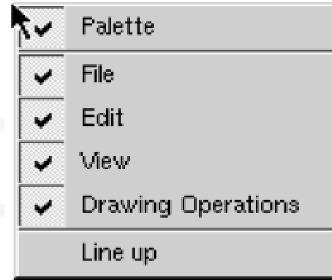


Şekil 6.12: Beş adet seyyar (dock) penceresi olan bir `QMainWindow`.

Yüzen bir seyyar pencerenin kapat düğmesine sahip olması arzu ediliyor ise bu durumda setCloseMode() fonksiyonu şu şekilde çağrılmalıdır:

```
dockWindow->setCloseMode(QDockWindow::Undocked);
```

QdockArea, bütün seyyar pencereleri ve alet kutuların listesini içeren bir siyakat menüsü (Şekil 6.13) tedarik eder. Seyyar pencere kapatıldıktan sonra kullanıcı bu siyakat menüsünü kullanarak onu açabilir.



Şekil 6.13: Bir `QDockArea` siyakat (context) menüsü.

Seyyar pencereler QdockWindow sınıfnın bir alt sınıfı olmak zorundadırlar. Eğer sadece düğmeler ve başka aletler içeren bir alet kutusuna ihtiyaç duyulursa bu durumda QToolBar sınıfı kullanılabilir ki bu QdockWindow un bir alt sınıfıdır. Şimdi bir QComboBox, bir QSpinBox ve bir takım düğmeleri içeren bir alet kutusunun (QToolBar) nasıl oluğturulabileceğini ve nasıl rıhtım alanına nasıl yerleştirilebileceğinizi göstereceğiz:

```
QToolBar *toolBar = new QToolBar(tr("Font"), this);
QComboBox *fontComboBox = new QComboBox(true, toolBar);
QSpinBox *fontSize = new QSpinBox(toolBar);
boldAct->addTo(toolBar);
italicAct->addTo(toolBar);
underlineAct->addTo(toolBar);
```

```
moveDockWindow(toolBar, DockBottom);
```

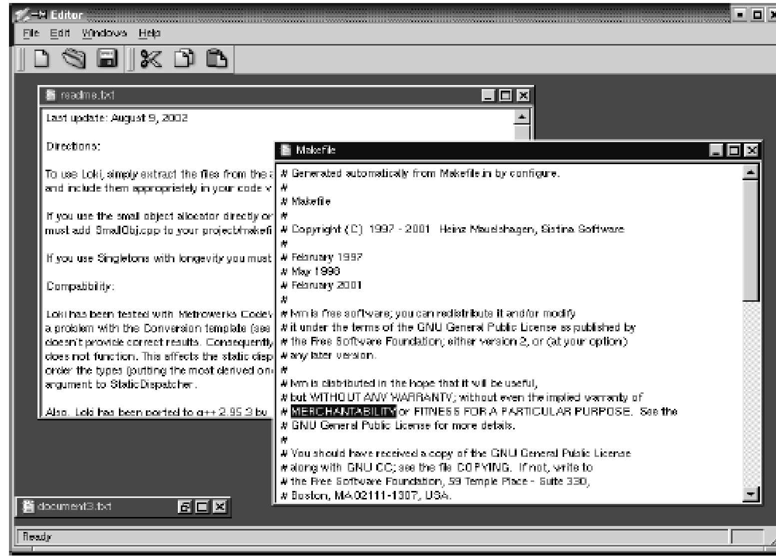
QComboBox ve QSpinBox aletlerinin genişlikleri normal bir alet kutusundan çok fazla olduğundan kullanıcı bu seyyar pencereyi QMainWindow un saü yada solundaki rıhtım alanına yerleştirimesi durumunda hoş bir görüntü arzetmez. Bunu önlemek için QmainWindow::setDockEnabled() fonksiyonunu şu şekilde çağırabiliriz:

```
setDockEnabled(toolBar, DockLeft, false);
setDockEnabled(toolBar, DockRight, false);
```

Eğer maksadımız yüzen bir alet yada bir alet paleti ise bu durumda QDockWindow kullanıp setWidget() fonksiyonunu çağırmak suretiyle istediğimiz aleti seyyar pencere içerisinde görüntüleyebiliriz. Bu alet istenildiği kadar karmaşık olabilir. Kullanıcınn seyyar pencerenin rıhtım alanında olduğu halde ebatını değiştirmesine izin vermek istersek bu durumda setResizeEnabled() fonksiyonunu söz konusu seyyar pencere için çağırmalıyız. Seyyar pencere haliyle ayırıcıya benzer bir kulpu olduğu halde görüntülenir. Aletin düşey yada yatay pozisyonda olmasına bağlı olarak şekil değiştirmesini arzu eder isek bu durumda QDockWindow:: setOrientation() fonksiyonunu yeniden tanımlamalıp gerekli değişikliği orada yapmalıyız. Bütün seyyar pencerelerin ve alet kutularının pozisyonlarını, proğramın daha sonra koşturulduğunda kullanmak üzere, muhafaza etmek için QSplitter in seçeneklerini kaydetmek için kullanmış oldığumuz (p. 143) koda benzer kod buradada kullanabiliriz. Özet olarak, bu kod kaydetme işlemi için QMainWindow sınıfının << operatörünü ve yüklemek için ise QmainWindow sınıfının >> operatörünü kullanır. Microsoft Visual Studio ve Qt Designer gibi proğramlar kullanıcıya asnek bir GUI sunmak için rıhtım alanını çokca kaullanırlarç. Qt altında bunu başarmak için çok sayıda hususi QdockWindow ları ve ortada MDI çocuk aletlerini kontrol etmek için bir QWorkspace ihtiva eden bir QMainWindow kullanılır.

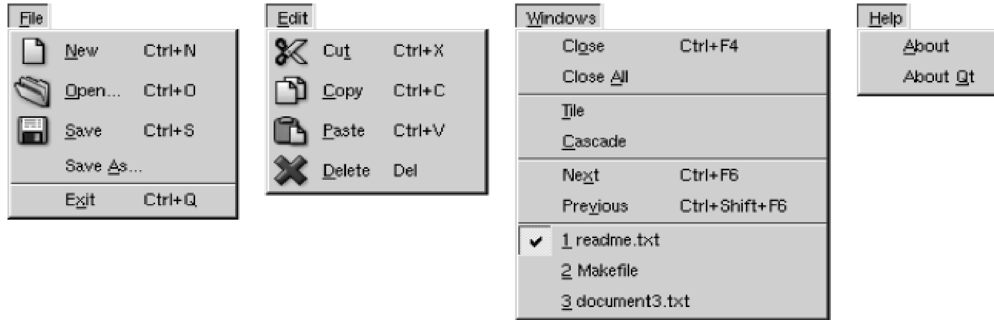
Müteaddid Doküman Arabirimi (MDI)

Ana pencerenın merkezi alanında birden fazla doküman iöerebilen proğramlara MDI (multiple document interface) adı verilir. At altında MDI proğramları oluşturmak için QWorkspace sınıfı merkezi alet olarak kullanılır ve doküman pencereleri QWorkspace in çocukları yapılır. Genelde MDI proğramları pencereleri tanzim etmek maksadıyla bir "Pencereler" menüsü tedarik ederler. Bu menü altında aktih olan pencere bir çek mark ile gösterilir. Kullanıcı "Pencereler" menüsü altında istediği pemcereyi seçerek aktif hale getirebilir. Bu kısımda, MDI propğramlarının nasıl yazıldığını ve Pencereler menüsünün nasıl oluşturulabileceğini göstermak maksadı ile, Şekil 6.14 de gösterilen Editor ptoğramını geliştireceğiz.



Şekil 6.14 : Editor proğramı.

Bu proğram MainWindow ve Editor namında iki sınıfdan ibarettir: Kaynak kodunu CD de bulabilirsiniz ve kodun çoğo daha önce görmüş olduğumuz Spreadsheed proğramının aynısı yada ona çok benzer.



Şekil 6.15 : Editor proğramının menüleri.

MainWindow sınıfı ile başlayalım.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
      : QMainWindow(parent, name)
{
      workspace = new QWorkspace(this);
      setCentralWidget(workspace);
      connect(workspace, SIGNAL(windowActivated(QWidget *)), this,
                                SLOT(updateMenus()));
      connect(workspace, SIGNAL(windowActivated(QWidget *)), this,
                                SLOT(updateModIndicator()));
      createActions();
      createMenus();
      createToolBars();
```

```
		createStatusBar();
		setCaption(tr("Editor"));
		setIcon(QPixmap::fromMimeSource("icon.png"));
	}
```

MainWindow un yapıcısında bir QWorkspace aleti oluşturup onu merkezi alet yapıyoruz. QWorkspace sınıfının windowActivated() sinyalini iki tane hususi dilime baplıyoruz. Bu dilimlerin vazifesi menülerin ve durum çubuğunun aktif olan pencere ila uyumlu olmasını sağlamaktır.

```
	void MainWindow::newFile()
	{
		Editor *editor = createEditor();
		editor->newFile();
		editor->show();
	}
```

newFile() dilimi Dosya|Yeni menüsüne tekabül eder. Bir çocuk Editor penceresi oluşturmak için createEditor() hususi fonksiyonundan yararlanır.

```
	Editor *MainWindow::createEditor()
	{
		Editor *editor = new Editor(workspace);
		connect(editor, SIGNAL(copyAvailable(bool)), this,
			SLOT (copyAvailable(bool)));
		connect(editor, SIGNAL(modificationChanged(bool)), this,
			SLOT(updateModIndicator()));
		return editor;
	}
```

createEditor() fonksiyonu Editor aletini oluşturur ve iki tane sinyal ve dilim bağlantısı kurar. Yapılan ilk bağlantının amacı Değiştir|Kes ve Değiştir|Kopyala menülerini seçilmiş bir metin olup olmamasına bağlı olarak ya muktedir yada malul kılar. İkinci bağlantı ise durum çubuğundaki DEĞ göstergesinin daima güncel olmasını sağlamaktır. MDI kullandığımız için birden fazla Editor aletinin aynı anda kullanılması muhtemeldir. Bunun ehemmiyet arzetmesinin sebebi bizim sadece aktif olan Editor penceresinin copyAvailable(bool) ve modificationChanged() sinyallerine cevap vermek istememizdir. Bu bir problem teşkil etmez çünküğ bu sinyaller ancak aktif pencere tarafından yayınlanabilir.

```
	void MainWindow::open()
	{
		Editor *editor = createEditor();
		if (editor->open())
			editor->show();
		else
			editor->close();
	}
```

open() fonksiyonu Dosya|Aç menüsüne tekabül eder. O yeni doküman için bir yeni Editor açar ve Editor un open() fonksiyonunu açar. Dosya operasyonlarının Editor sınıfı içerisinde gerçekleştirmek MainWindow içerisinde gerçekleştirmekten daha makuldür çünkü her Editor kendi bağımsızlığını muhafaza etmelidir. Açma işleminin başarısızlıkla sonuçlanması durumunda, kullanıcının zaten bu başarısızlıktan haberdar edilmiş olması sebebiyle, editörü

kapatıyoruz.

```
void MainWindow::save()
{
      if (activeEditor())
      {
            activeEditor()->save();
            updateModIndicator();
      }
}
```

save() dilimi, varsa, aktif olan editörün save() fonksiyonunu çağırır. Buradada asıl kaydetme işini yapan kod Editor sınınının save fonksiyonunda yer almaktadır.

```
Editor *MainWindow::activeEditor()
{
      return (Editor *)workspace->activeWindow();
}
```

activeEditor() hususi fonksiyonu aktif olan çocuk pencerenin bir müşirini Editor türünden verir.

```
void MainWindow::cut()
{
      if (activeEditor())
            activeEditor()->cut();
}
```

cut() dilimi editörün cut() fonksiyonunu çağırır. copy(), paste() ve del() dilimleride aynı yolu takip ederler.

```
void MainWindow::updateMenus()
{
      bool hasEditor = (activeEditor() != 0);
      saveAct->setEnabled(hasEditor);
      saveAsAct->setEnabled(hasEditor);
      pasteAct->setEnabled(hasEditor);
      deleteAct->setEnabled(hasEditor);
      copyAvailable(activeEditor()
            && activeEditor()->hasSelectedText());
      closeAct->setEnabled(hasEditor);
      closeAllAct->setEnabled(hasEditor);
      tileAct->setEnabled(hasEditor);
      cascadeAct->setEnabled(hasEditor);
      nextAct->setEnabled(hasEditor);
      previousAct->setEnabled(hasEditor);

      windowsMenu->clear();
      createWindowsMenu();
}
```

updateMenus() dilimi ne zamanki bi pencere aktif hale getirilirse (veya en son pencere kapatılırsa) menü sistemini güncelleştirmek için MainWindow yağıcısı içerisine yerleştirmiş olduğumuz sinyaş dilim mekanizaması sayesınde çağrılır. Çoğu menü seçenekleri yanlız aktif bir pencerenin olması durumunda kullanılabilirler bundan dolayı onları aktif

pencerenin olmaması durumunda malul yapıyoruz. Daha sonra “Pencereler” menüsüsnün içeriğini silip createWindowsMenu() fonksiyonunu çağırmak suretiyle pencereler listesini güncelleştiriyoruz.

```
void MainWindow::createWindowsMenu()
{
   closeAct->addTo(windowsMenu);
   closeAllAct->addTo(windowsMenu);
   windowsMenu->insertSeparator();
   tileAct->addTo(windowsMenu);
   cascadeAct->addTo(windowsMenu);
   windowsMenu->insertSeparator();
   nextAct->addTo(windowsMenu);
   previousAct->addTo(windowsMenu);
   if (activeEditor())
   {
         windowsMenu->insertSeparator();
         windows = workspace->windowList();
         int numVisibleEditors = 0;
         for (int i = 0; i < (int)windows.count(); ++i)
         {
               QWidget *win = windows.at(i);
               if (!win->isHidden())
               {
                     QString text = tr("%1 %2")
                           .arg(numVisibleEditors + 1)
                           .arg(win->caption());
                     if (numVisibleEditors < 9)
                           text.prepend("&");
                     int id = windowsMenu->insertItem( text, this,
                                  SLOT(activateWindow(int)));
                     bool isActive = (activeEditor() == win);
                     windowsMenu->setItemChecked(id, isActive);
                     windowsMenu->setItemParameter(id, i);
                     ++numVisibleEditors;
               }
         }
   }
}
```

createWindowsMenu() hususi fonksiyonu “Pencereler” menüsünü görülebilir pencerelerin listesi ve bir takım actions ile doldurur. Sözkonusu actions genellikle bu tür menülerde kullanılırlar ve QWorkspace sınıfının closeActiveWindow(), closeAllWindows(), tile() ve cascade() dilimlerini kullanarak kolayca oluşturulabilirler. Pencereler menüsü altında aktif olan pencerenin yanında çek mark görüntülenmektedir. Kullanıcı bir pencereyi seçtiğinde activateWindow() dilimi o pencerenin indeksi ile çağrılır bunun sebebi ise setItemParameter() fonksiyonudur. Bu üçüncü bölümde yazmış olduğumuz Spreadsheet proğramının son açılan dosyalar listesini (p. 54) oluştururken takip ettiğimiz metoda çok benzer. Ilk dokuz pencerenin numarasaının hemen önüne "&" koymak suretiyle birden dokuza kadar olan rakamları kısa yol anahtarı yapıyoruz. Menü altındaki diğer girdiler için

kısa yol anahtarı oluşturmuyoruz.

```
void MainWindow::activateWindow(int param)
{
      QWidget *win = windows.at(param);
      win->show();
      win->setFocus();
}
```

activateWindow() fonksiyonu Pencereler menüsünden her hangi bir pencerenin seçilmesi durumunda çağrılır. Bu fonksiyonun int türündeki parametresi setItemParameter() fonksiyonu ile ayarlanan değerdir. “windows” üye deüişkeni pencerelerin bir listesini ihtiva eder ve createWindowsMenu() fonksiyonu içerisinde doldurulmuştur.

```
void MainWindow::copyAvailable(bool available)
{
cutAct->setEnabled(available);
copyAct->setEnabled(available);
}
```

copyAvailable() dilimi editör içerisinde bir metnin seçilmesi veya seçilmesinden vazgeöilmesi durumunda çağrılır. Kes ve Kopyala fiillerini muktedir veya malul kılar.

```
void MainWindow::updateModIndicator()
{
      if (activeEditor() && activeEditor()->isModified())
            modLabel->setText(tr("MOD"));
      else
            modLabel->clear();
}
```

updateModIndicator() dilimi durum çubuğundaki DEĞ göstergesini güncelleştirir. Bu dilim editördeki metnin değişmesi durumunda çağrılır. Bu dilim yeni bir pencere aktif hale getirildiğindede çağrılır.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
      workspace->closeAllWindows();
      if (activeEditor())
            event->ignore();
      else
            event->accept();
}
```

closeEvent() fonksiyonu bütün çocuk pencereleri kapatması için yeniden tanımlandı. Çocuk aletlerden birisi kapat eylemine (close event) iltifat etmemesi durumunda (kullanıcının kaydedilmemiş değişiklikleri ihbar eden mesaj kutusunun vazgeç düğmesine basması gibi durumlarda), MainWindow için yayınlanan kapat eylemini reddediyoruz; aksi takdirde kapatma eylemini kabul edip Qt nin pencereyi kapatmasına neden oluyoruz. Eğer closeEvent() fonksiyonunu MainWindow içerisinde yeniden tanımlamış olmasaydık bu durumda kullanıcının kaydedilmemiş değişiklikleri kaydetmesine fırsat verilmemiş olurdu. Böylece MainWindow un kodunu gözden geçirmiş olduk ve şimdi Editor sınıfına geçebiliriz. Editor sınıfı bir çocuk pencereyi teşkil eder. O QtextEdit in bir alt sınıfıdır ve metin edit

etme yeteneğine sahiptir. Her bir Qt aleti yanlız başına bir pencere olarak kullanılabileceği gibi MDI içerisindede bir çocuk pencere olarakta kullanılabilir. İşte Editor sınınfının tanımı:

```
class Editor : public QTextEdit
{
      Q_OBJECT
public:
      Editor(QWidget *parent = 0, const char *name = 0);
      void newFile(); bool open();
      bool openFile(const QString &fileName);
      bool save();
      bool saveAs();
      QSize sizeHint() const;
signals:
      void message(const QString &fileName, int delay);
protected:
      void closeEvent(QCloseEvent *event);
private:
      bool maybeSave();
      void saveFile(const QString &fileName);
      void setCurrentFile(const QString &fileName);
      QString strippedName(const QString &fullFileName);
      bool readFile(const QString &fileName);
      bool writeFile(const QString &fileName);
      QString curFile;
      bool isUntitled;
      QString fileFilters;
};
```

Spreadsheet proğramının MainWindow sınıfının hususi fonksiyonlarından dört tanesi Editor sınıfındada mevcuttur: maybeSave(), saveFile(), setCurrentFile(), ve strippedName().

```
Editor::Editor(QWidget *parent, const char *name)
      : QTextEdit(parent, name)
{
      setWFlags(WDestructiveClose);
      setIcon(QPixmap::fromMimeSource("document.png"));
      isUntitled = true;
      fileFilters = tr("Text files (*.txt)\n" "All files (*)");
}
```

Editor sınıfının yapıcısı WDestructiveClose seçeneğini setWFlags() fonksiyonunu çağırarak ayarlar. Bri sınıfın yapıcısı "flags[4]" parametresi tedarik etmiyor ise (QtextEdit sınıfında olduğu gibi) bu durumda çoğu seçenekleri setWFlags() fonksiyonunu kullanarak ayarlayabiliriz.

Kullanıcının arzu ettiği sayıda pencere açmasına izin verdiğimizden dolayı bu pencerelerin kaydedilmelerinden önce birbirlerinden ayırt edilebilmeleri için bir ismlendirme sistemi kullanmamız gerekiyor. Sıkça kullanılab bir metog isimlerin numaralandırılmasıdır (mesela, document1.txt). `isUntitled` değişkeninden yararlanarak kullanıcının vermiş olduğu isimleri bizim otomatik olarak vemiş olduğumuz isimlerden ayırdediyoruz. Yapıcıdan sonra

4 seçenekler, ayarlar (lafzen bayraklar anlamına gelir)

ya newFile() veya open() fonksiyonlarının çağrılmasını bekliyoruz.

```
void Editor::newFile()
{
      static int documentNumber = 1;
      curFile = tr("document%1.txt").arg(documentNumber);
      setCaption(curFile);
      isUntitled = true;
      ++documentNumber;
}
```

newFile() fonksiyonu yeni doküman için document2.txt gibi isim oluşturur. Bu işlemi yapan kodu yapıcı yerine newFile() içine yerleştirmemizin sebebi kullanıcı var olan bir dosyayı açtığında boş yere isim oluşturmamaktır. documentNumber değişkeni static türünden olduğundan bu Editor sınıfının her bir nesnesi tarafından paylaşılmaktadır.

```
bool Editor::open()
{
      QString fileName =
            QFileDialog::getOpenFileName(".", fileFilters, this);
      if (fileName.isEmpty())
            return false;
      return openFile(fileName);
}
```

open() fonksiyonu openFile() fonksiyonunu kullanarak var olan bir dosyayı açar.

```
bool Editor::save()
{
      if (isUntitled)
      {
            return saveAs();
      }
      else
      {
            saveFile(curFile);
            return true;
      }
}
```

save() fonksiyonu isUntitled değişkeninden yararlanarak ya saveFile() yada saveAs() fonksiyonunu çağırır.

```
void Editor::closeEvent(QCloseEvent *event)
{
      if (maybeSave())
            event->accept();
      else
            event->ignore();
}
```

closeEvent() fonksiyonu kullanıcının kaydedilmeiş değişiklikleri kaydetmesine imkan vermek için yeniden tanımlandı. Asıl işi yapan kod maybeSave() fonksiyonunda yer almaktadır ki bu fonksiyon bir mesaj kutusu açıp kullanıcnın değişiklikleri kaydedip istemediğini sorar. Eşer maybeSave() fonksiyonu müsbet geri getiri ise biz kapat eylemini

kabul ederiz aksi takdirde bu eylemi reddedip pencereyi açık bırakırız.

```
void Editor::setCurrentFile(const QString &fileName)
{
      curFile = fileName;
      setCaption(strippedName(curFile));
      isUntitled = false;
      setModified(false);
}
```

setCurrentFile() fonksiyonu openFile() ve saveFile() fonksiyonları tarafından curFile ve isUntitled değişkenlerini güncelleştirmek, pencerenin başlığını tesbit etmek, ve editörün “modified” seçeneğini menfi yapmak için çağrılır. Editor sınıfı setModified() ve isModified() fonksiyonlarını QtextEdit den miras aldığından kendine has bir modified seçeneği tedarik etmesine gerek yoktur. Kullanıcı editördeki metni değiştirir değiştirmez QTextEdit modificationChanged() sinyalini yayınlar ve batni (internal) “modifed” seçeneğini müsbet yapar.

```
QSize Editor::sizeHint() const
{
return QSize(72 * fontMetrics().width( x ),
            25 * fontMetrics().lineSpacing());
}
```

sizeHint() fonksiyonu “x” harfinin ebatına ve bir satırın yüksekliğine bağlı olarak ideal boyutu verir. QWorkspace ise bu ideal boyutu kullanarak pencereye ilk boyutunu verir. Nihayet işte Editor proğramının main.cpp dosyası:

```
#include <qapplication.h>
#include "mainwindow.h"
int main(int argc, char *argv[])
{
      QApplication app(argc, argv);
      MainWindow mainWin;
      app.setMainWidget(&mainWin);
      if (argc > 1)
      {
            for (int i = 1; i < argc; ++i)
                  mainWin.openFile(argv[i]);
      }
      else
      {
            mainWin.newFile();
      }
      mainWin.show();
      return app.exec();
}
```

Kullanıcı komut satırında herhangi bir doya verir ise biz onu açmaya teşebbüs ederiz. Aksi takdirde proğramı boğ bir dosya ile başlatırız. Qt ye has -style, -font gibi komut satırı seçenekleri QApplication nın yapıcısı tarafından argüman listesinden uzaklaştırılırlar. Yani aşağıdaki komut satırına

```
editor -style=motif readme.txt
```

yazıp enter tuşuna bastığımızda Editor proğramı readme.txt dokümanını açar. Çok sayıda dokümanı aynı anda açık tutmanın bir yolu MDI metodudur. Bir diğer yaklaşımda müteaddid üst seviye pencereler kullanmaktadır. Bu yaklağım üçüncü bölümde ele alındı.